BOOK 19: TYPESCRIPT BASICS
Chapter 1: What is TypeScript

TYPESCRIPT EXPLAINED

TypeScript is JavaScript with types.

It adds type checking to JavaScript.

You write TypeScript and it compiles to JavaScript.

Browsers and Node run the compiled JavaScript.

WHY TYPESCRIPT EXISTS

JavaScript has no type checking.

You can add a number to a string.

You can call methods that do not exist.

Errors happen at runtime when users see them.

TypeScript catches these errors while you code.

THE PROBLEM TYPESCRIPT SOLVES

This JavaScript code runs but fails:

```
let user = { name: "Ali", age: 25 }
console.log(user.email.toUpperCase())
```

JavaScript runs this.

It crashes because email is undefined.

User sees the error.

TypeScript catches this before running:

```
let user = { name: "Ali", age: 25 }
console.log(user.email.toUpperCase())
```

TypeScript error: Property email does not exist on type.

You fix it before anyone sees the bug.

STATIC VS DYNAMIC TYPING

Dynamic typing (JavaScript):

Types checked while program runs.

Errors happen at runtime.

More flexible but more bugs.

Static typing (TypeScript):

Types checked before program runs.

Errors caught during development.

More structure but fewer bugs.

TYPESCRIPT COMPILES TO JAVASCRIPT

TypeScript code:

```
let message: string = "Hello"
console.log(message)
```

Compiles to JavaScript:

```
var message = "Hello";
console.log(message);
```

Types are removed.

Plain JavaScript remains.

INSTALLING TYPESCRIPT

Global installation:

```
npm install -g typescript
```

Now tsc command is available.

Check version:

```
tsc --version
```

Project installation:

npm init -y
npm install typescript --save-dev

Better for team projects.

YOUR FIRST TYPESCRIPT FILE

Create file hello.ts:

```
let greeting: string = "Hello TypeScript"
console.log(greeting)
```

Compile to JavaScript:

tsc hello.ts

Creates hello.js file.

Run the JavaScript:

node hello.js

Output: Hello TypeScript

TYPESCRIPT CONFIGURATION

Create tsconfig.json:

tsc --init

Creates configuration file with many options.

Basic tsconfig.json:

```
{
 "compilerOptions": {
 "target": "ES2020",
 "module": "commonjs",
 "strict": true,
 "outDir": "./dist",
 "rootDir": "./src"
 },
 "include": ["src/**/*"]
}
```

target - JavaScript version to compile to.

module - Module system to use.

strict - Enable all strict type checks.

outDir - Where compiled files go.

rootDir - Where source files are.

COMPILING WITH CONFIG

Compile all files:

tsc

Uses tsconfig.json settings.

Watch mode (recompile on change):

tsc --watch

Or:

tsc -w

TYPE INFERENCE

TypeScript guesses types when possible:

let name = "Ali"

TypeScript knows name is string.

You did not write the type.

This still causes error:

let name = "Ali"
name = 42

Cannot assign number to string.

EXPLICIT TYPES

Add types yourself:

let name: string = "Ali"
let age: number = 25
let active: boolean = true

Colon followed by type name.

TYPE ERRORS

TypeScript shows errors in editor:

let age: number = "twenty five"

Red underline appears.

Error: Type string is not assignable to type number.

BENEFITS OF TYPESCRIPT

Catch bugs early:

Errors found while coding not while running.

Better autocomplete:

Editor knows what methods exist.

Self documenting code:

Types explain what data looks like.

Safer refactoring:

Change code and see what breaks.

Team communication:

Types show how to use functions.

TYPESCRIPT VS JAVASCRIPT

When to use JavaScript:

Small scripts.

Quick prototypes.

Learning programming.

When to use TypeScript:

Large applications.

Team projects.

Long-term maintenance.

Complex data structures.

TYPESCRIPT ECOSYSTEM

Editor support:

VS Code has best TypeScript support.

Built-in type checking.

Autocomplete and refactoring.

Type definitions:

Libraries need type information.

DefinitelyTyped provides types.

Install with npm install @types/libraryname.

Frameworks:

React works great with TypeScript.

Node.js works with TypeScript.

Express has type definitions.

DEVELOPMENT WORKFLOW

1. Write TypeScript code.

2. TypeScript checks for errors.

3. Fix errors in editor.

4. Compile to JavaScript.

5. Run JavaScript.

With watch mode:

1. Run tsc --watch.

2. Write TypeScript.

3. Save file.

4. Automatic compile.

5. See errors immediately.

PRACTICE

Install TypeScript globally.

Create a hello.ts file.

Add a typed variable.

Compile with tsc.

Run with node.

Create a tsconfig.json.

WHAT YOU LEARNED

You learned what TypeScript is.

You learned why types matter.

You learned to install TypeScript.

You learned to compile TypeScript.

You learned about configuration.

The next chapter teaches basic types.
BOOK 19: TYPESCRIPT BASICS
Chapter 2: Basic Types

PRIMITIVE TYPES

TypeScript has the same primitives as JavaScript.

string, number, boolean.

Plus additional types for safety.

STRING TYPE

Text values:

```
let name: string = "Ali"
let greeting: string = "Hello"
let message: string = `Welcome ${name}`
```

Template literals work.

Single or double quotes work.

NUMBER TYPE

All numbers (integer and decimal):

```
let age: number = 25
let price: number = 19.99
let negative: number = -10
let hex: number = 0xff
let binary: number = 0b1010
let octal: number = 0o744
```

JavaScript has one number type.

No separate int or float.

BOOLEAN TYPE

True or false:

```
let isActive: boolean = true
let isComplete: boolean = false
let hasAccess: boolean = true
```

NULL AND UNDEFINED

Absence of value:

```
let nothing: null = null
let notDefined: undefined = undefined
```

Usually used with other types.

ANY TYPE

Disables type checking:

```
let data: any = "hello"
data = 42
data = true
data = { name: "Ali" }
```

Any value allowed.

No type errors.

Defeats purpose of TypeScript.

Avoid when possible.

UNKNOWN TYPE

Safer than any:

```
let data: unknown = "hello"
data = 42
```

Can assign anything.

But cannot use without checking:

```
let data: unknown = "hello"
console.log(data.toUpperCase())
```

Error: Object is of type unknown.

Must narrow the type first:

```
let data: unknown = "hello"
```

```
if (typeof data === "string") {
 console.log(data.toUpperCase())
}
```

Works after type check.

VOID TYPE

Function returns nothing:

```
function logMessage(message: string): void {
 console.log(message)
}
```

No return statement.

Or empty return.

NEVER TYPE

Function never returns:

```
function throwError(message: string): never {
 throw new Error(message)
}

function infiniteLoop(): never {
 while (true) {
 console.log("running")
 }
}
```

Function crashes or loops forever.

ARRAY TYPES

Two syntaxes:

```
let numbers: number[] = [1, 2, 3, 4, 5]
let names: string[] = ["Ali", "Sara", "Omar"]
```

Generic syntax:

```
let numbers: Array<number> = [1, 2, 3, 4, 5]
let names: Array<string> = ["Ali", "Sara", "Omar"]
```

Both work the same.

First syntax is more common.

TUPLE TYPES

Fixed length array with specific types:

```
let person: [string, number] = ["Ali", 25]
```

First element must be string.

Second element must be number.

Access by index:

```
let name = person[0]
let age = person[1]
```

Wrong order causes error:

let person: [string, number] = [25, "Ali"]

Error: Type number is not assignable to type string.

ENUM TYPES

Named constants:

```
enum Direction {
 Up,
 Down,
 Left,
 Right
}
```

let move: Direction = Direction.Up

Values start at 0 by default.

Custom values:

```
enum Status {
 Pending = 1,
 Active = 2,
 Completed = 3
}
```

String enums:

```
enum Color {
 Red = "RED",
 Green = "GREEN",
 Blue = "BLUE"
```

}

let favorite: Color = Color.Blue

OBJECT TYPE

Generic object:

let user: object = { name: "Ali", age: 25 }

Not very useful.

Cannot access properties.

Better to use specific shape:

```
let user: { name: string; age: number } = {
 name: "Ali",
 age: 25
}
```

Now TypeScript knows the properties.

LITERAL TYPES

Exact values:

```
let direction: "up" | "down" = "up"
direction = "down"
direction = "left"
```

Error: left is not assignable.

Number literals:

```
let dice: 1 | 2 | 3 | 4 | 5 | 6 = 1
dice = 7
```

Error: 7 is not assignable.

UNION TYPES

Multiple possible types:

```
let id: string | number = "abc123"
id = 123
```

Either string or number allowed.

Common use case:

```
function printId(id: string | number) {
 console.log("ID: " + id)
}

printId("abc")
printId(123)
```

TYPE INFERENCE

TypeScript figures out types:

```
let name = "Ali"
```

TypeScript infers string.

let numbers = [1, 2, 3]

TypeScript infers number array.

let user = { name: "Ali", age: 25 }

TypeScript infers object shape.

WHEN TO ADD TYPES

Always add types for:

Function parameters.

Function return types.

Variables initialized later.

Can skip types for:

Variables initialized immediately.

Return types with obvious returns.

Example:

let name = "Ali"

```
function greet(name: string): string {
 return "Hello " + name
}
```

TYPE ASSERTIONS

Tell TypeScript what type something is:

```
let input = document.getElementById("username") as
HTMLInputElement
input.value = "Ali"
```

Alternative syntax:

```
let input =
<HTMLInputElement>document.getElementById("username")
```

First syntax preferred (works in JSX).

Use when you know more than TypeScript.

PRACTICE

Declare variables with each basic type.

Create an array of numbers.

Create a tuple with string and number.

Define an enum for days of week.

Use union types for a flexible variable.

Practice type inference.

WHAT YOU LEARNED

You learned primitive types.

You learned any, unknown, void, never.

You learned arrays and tuples.

You learned enums and literals.

You learned union types and assertions.

The next chapter teaches type annotations.

BOOK 19: TYPESCRIPT BASICS

Chapter 3: Type Annotations

WHAT ARE TYPE ANNOTATIONS

Type annotations explicitly declare types.

You tell TypeScript what type something is.

Uses colon followed by type.

VARIABLE ANNOTATIONS

Add type after variable name:

```
let name: string = "Ali"
let age: number = 25
```

let active: boolean = true

Colon then type name.

WHEN TO ANNOTATE VARIABLES

Annotate when:

Variable declared without value.

```
let username: string
username = getUserInput()
```

Type is not obvious.

```
let data: string | null = getData()
```

Can skip when:

Value assigned immediately.

```
let name = "Ali"
```

TypeScript infers string.

FUNCTION PARAMETER ANNOTATIONS

Parameters need types:

```
function greet(name: string) {
 console.log("Hello " + name)
```

```
}

greet("Ali")
greet(123)
```

Error: Argument 123 is not assignable to string.

MULTIPLE PARAMETERS

Each parameter gets its type:

```
function add(a: number, b: number) {
 return a + b
}

add(5, 3)
add("5", "3")
```

Error: String not assignable to number.

RETURN TYPE ANNOTATIONS

Add after parameter list:

```
function add(a: number, b: number): number {
 return a + b
}
```

Colon after parentheses.

Then return type.

VOID RETURN TYPE

Function returns nothing:

```
function logMessage(message: string): void {
 console.log(message)
}
```

No return statement needed.

OPTIONAL PARAMETERS

Add question mark:

```
function greet(name: string, greeting?: string) {
 if (greeting) {
 console.log(greeting + " " + name)
 } else {
 console.log("Hello " + name)
 }
}

greet("Ali")
greet("Ali", "Welcome")
```

Both calls work.

Optional parameters can be undefined.

DEFAULT PARAMETERS

Provide default value:

```
function greet(name: string, greeting: string = "Hello") {
 console.log(greeting + " " + name)
}

greet("Ali")
greet("Ali", "Welcome")
```

Default implies optional.

Type inferred from default value.

OBJECT ANNOTATIONS

Describe object shape:

```
let user: { name: string; age: number } = {
 name: "Ali",
 age: 25
}
```

Semicolons between properties.

Or commas work too.

NESTED OBJECTS

Objects inside objects:

```
let user: {
 name: string;
 address: {
 city: string;
 country: string;
 };
```

```
} = {
 name: "Ali",
 address: {
 city: "Cairo",
 country: "Egypt"
 }
}
```

OPTIONAL PROPERTIES

Properties that may not exist:

```
let user: {
 name: string;
 age: number;
 email?: string;
} = {
 name: "Ali",
 age: 25
}
```

email is optional.

Can be present or absent.

READONLY PROPERTIES

Properties that cannot change:

```
let user: {
 readonly id: number;
 name: string;
} = {
 id: 1,
```

```
 name: "Ali"
}

user.name = "Omar"
user.id = 2
```

Error: Cannot assign to id because it is readonly.

ARRAY ANNOTATIONS

Simple arrays:

```
let numbers: number[] = [1, 2, 3]
let names: string[] = ["Ali", "Sara"]
```

Arrays of objects:

```
let users: { name: string; age: number }[] = [
 { name: "Ali", age: 25 },
 { name: "Sara", age: 30 }
]
```

FUNCTION TYPE ANNOTATIONS

Variable that holds a function:

```
let greet: (name: string) => string

greet = function(name) {
 return "Hello " + name
}
```

Arrow syntax for function type.

Parameters in parentheses.

Arrow then return type.

CALLBACK ANNOTATIONS

Function parameter that is a function:

```
function processUser(
 user: { name: string },
 callback: (greeting: string) = > void
) {
 let greeting = "Hello " + user.name
 callback(greeting)
}

processUser(
 { name: "Ali" },
 function(message) {
 console.log(message)
 }
)
```

REST PARAMETERS

Multiple arguments as array:

```
function sum(...numbers: number[]): number {
 let total = 0
 for (let num of numbers) {
 total + = num
 }
```

```
 return total
}

sum(1, 2, 3, 4, 5)
```

Rest parameter is array type.

DESTRUCTURED PARAMETERS

Object destructuring:

```
function greet({ name, age }: { name: string; age: number }) {
 console.log(name + " is " + age)
}

greet({ name: "Ali", age: 25 })
```

Array destructuring:

```
function getFirst([first, second]: [string, string]): string {
 return first
}

getFirst(["Ali", "Sara"])
```

THIS PARAMETER

Specify type of this:

```
interface User {
 name: string;
 greet(this: User): void;
```

```
}

let user: User = {
 name: "Ali",
 greet() {
 console.log("Hello " + this.name)
 }
}
```

this parameter is not a real parameter.

Just tells TypeScript what this is.

OVERLOAD ANNOTATIONS

Multiple function signatures:

```
function combine(a: string, b: string): string
function combine(a: number, b: number): number
function combine(a: any, b: any): any {
 return a + b
}

let result1 = combine("Hello", "World")
let result2 = combine(5, 3)
```

First two lines are overload signatures.

Last is implementation.

INDEX SIGNATURES

Objects with dynamic keys:

let scores: { [name: string]: number } = {}

scores["Ali"] = 95
scores["Sara"] = 88

Key type in brackets.

Value type after colon.

ANNOTATION BEST PRACTICES

Be specific:

let user: { name: string; age: number }

Not:

let user: object

Use inference when clear:

let name = "Ali"

Not:

let name: string = "Ali"

Always annotate function parameters:

function greet(name: string)

Annotate complex return types:

function getUser(): { name: string; age: number }

PRACTICE

Annotate function parameters.

Add return types to functions.

Create objects with specific shapes.

Use optional and readonly properties.

Define callback function types.

Practice destructured parameters.

WHAT YOU LEARNED

You learned variable annotations.

You learned function annotations.

You learned object annotations.

You learned optional and readonly.

You learned function types and callbacks.

The next chapter teaches interfaces.
BOOK 19: TYPESCRIPT BASICS
Chapter 4: Interfaces

WHAT IS AN INTERFACE

An interface defines the shape of an object.

It describes what properties and methods exist.

Like a contract for objects to follow.

BASIC INTERFACE

Define an interface:

```
interface User {
 name: string;
 age: number;
}
```

Use the interface:

```
let user: User = {
 name: "Ali",
 age: 25
}
```

WHY USE INTERFACES

Without interface:

```
let user1: { name: string; age: number } = { name: "Ali", age: 25 }
let user2: { name: string; age: number } = { name: "Sara", age: 30 }
```

Repeat the shape every time.

With interface:

```
interface User {
 name: string;
 age: number;
}

let user1: User = { name: "Ali", age: 25 }
let user2: User = { name: "Sara", age: 30 }
```

Define once, use everywhere.

OPTIONAL PROPERTIES

Properties that may not exist:

```
interface User {
 name: string;
 age: number;
 email?: string;
}

let user1: User = { name: "Ali", age: 25 }
let user2: User = { name: "Sara", age: 30, email:
"sara@example.com" }
```

Both are valid.

READONLY PROPERTIES

Properties that cannot change:

```
interface User {
 readonly id: number;
 name: string;
}

let user: User = { id: 1, name: "Ali" }
user.name = "Omar"
user.id = 2
```

Error: Cannot assign to id.

METHOD DEFINITIONS

Functions in interfaces:

```
interface User {
 name: string;
 greet(): string;
}

let user: User = {
 name: "Ali",
 greet() {
 return "Hello, I am " + this.name
 }
}
```

With parameters:

```
interface Calculator {
 add(a: number, b: number): number;
 subtract(a: number, b: number): number;
}
```

EXTENDING INTERFACES

Build on existing interfaces:

```
interface Person {
 name: string;
 age: number;
}

interface Employee extends Person {
 employeeId: number;
 department: string;
}

let employee: Employee = {
 name: "Ali",
 age: 25,
 employeeId: 12345,
 department: "Engineering"
}
```

Employee has all Person properties plus its own.

MULTIPLE INHERITANCE

Extend multiple interfaces:

```
interface Named {
 name: string;
}

interface Aged {
 age: number;
}
```

```
interface Person extends Named, Aged {
 email: string;
}

let person: Person = {
 name: "Ali",
 age: 25,
 email: "ali@example.com"
}
```

INTERFACE FOR FUNCTIONS

Define function shape:

```
interface GreetFunction {
 (name: string): string;
}

let greet: GreetFunction = function(name) {
 return "Hello " + name
}
```

INTERFACE FOR ARRAYS

Define array structure:

```
interface StringArray {
 [index: number]: string;
}

let names: StringArray = ["Ali", "Sara", "Omar"]
let first: string = names[0]
```

INDEX SIGNATURES

Objects with dynamic keys:

```
interface Scores {
 [studentName: string]: number;
}

let scores: Scores = {
 Ali: 95,
 Sara: 88,
 Omar: 92
}

scores["Fatima"] = 90
```

COMBINING FIXED AND DYNAMIC

Some known properties, some dynamic:

```
interface Config {
 name: string;
 version: number;
 [key: string]: string | number;
}

let config: Config = {
 name: "MyApp",
 version: 1,
 author: "Ali",
 port: 3000
}
```

INTERFACE MERGING

Same interface name combines:

```
interface User {
 name: string;
}

interface User {
 age: number;
}

let user: User = {
 name: "Ali",
 age: 25
}
```

Both properties required.

Useful for extending library types.

IMPLEMENTING INTERFACES

Classes can implement interfaces:

```
interface Animal {
 name: string;
 speak(): void;
}

class Dog implements Animal {
 name: string;

 constructor(name: string) {
 this.name = name
 }
```

```
 speak() {
 console.log(this.name + " barks")
 }
}

let dog = new Dog("Rex")
dog.speak()
```

Class must have all interface members.

MULTIPLE INTERFACE IMPLEMENTATION

Class implements many interfaces:

```
interface Walkable {
 walk(): void;
}

interface Swimmable {
 swim(): void;
}

class Duck implements Walkable, Swimmable {
 walk() {
 console.log("Duck walks")
 }

 swim() {
 console.log("Duck swims")
 }
}
```

INTERFACE VS TYPE ALIAS

Interface:

```
interface User {
 name: string;
}
```

Type alias:

```
type User = {
 name: string;
}
```

Differences:

Interface can be extended with extends.

Interface can be merged (same name).

Type alias can use unions and intersections.

When to use:

Use interface for object shapes.

Use type for unions, primitives, tuples.

HYBRID TYPES

Object that is also callable:

```
interface Counter {
 (start: number): string;
 interval: number;
 reset(): void;
```

```
}

function createCounter(): Counter {
 let counter = function(start: number) {
 return "Started at " + start
 } as Counter

 counter.interval = 1000
 counter.reset = function() {
 console.log("Reset")
 }

 return counter
}

let myCounter = createCounter()
myCounter(10)
myCounter.interval = 500
myCounter.reset()
```

GENERIC INTERFACES

Interfaces with type parameters:

```
interface Container<T> {
 value: T;
 getValue(): T;
}

let numberContainer: Container<number> = {
 value: 42,
 getValue() {
 return this.value
 }
}

let stringContainer: Container<string> = {
```

```
 value: "Hello",
 getValue() {
 return this.value
 }
}
```

COMMON INTERFACE PATTERNS

API Response:

```
interface ApiResponse<T> {
 data: T;
 status: number;
 message: string;
}
```

Event Handler:

```
interface EventHandler {
 (event: Event): void;
}
```

Configuration:

```
interface Config {
 readonly apiUrl: string;
 timeout?: number;
 retries: number;
}
```

PRACTICE

Create an interface for a product.

Add optional and readonly properties.

Extend an interface.

Implement an interface with a class.

Create a generic interface.

Use index signatures.

WHAT YOU LEARNED

You learned to define interfaces.

You learned optional and readonly properties.

You learned to extend interfaces.

You learned to implement interfaces.

You learned generic interfaces.

The next chapter teaches type aliases.

BOOK 19: TYPESCRIPT BASICS
Chapter 5: Type Aliases

WHAT IS A TYPE ALIAS

A type alias gives a name to a type.

Create once, use everywhere.

Makes complex types readable.

BASIC TYPE ALIAS

Create an alias:

```
type ID = string

let userId: ID = "abc123"
let productId: ID = "xyz789"
```

ID is now a name for string.

OBJECT TYPE ALIAS

Name an object shape:

```
type User = {
 name: string;
 age: number;
 email: string;
}

let user: User = {
 name: "Ali",
 age: 25,
 email: "ali@example.com"
}
```

Similar to interface but different syntax.

UNION TYPES

Combine multiple types:

```
type StringOrNumber = string | number

let id: StringOrNumber = "abc123"
id = 123
```

Either type is allowed.

PRACTICAL UNION TYPES

Status values:

```
type Status = "pending" | "active" | "completed"

let orderStatus: Status = "pending"
orderStatus = "active"
orderStatus = "cancelled"
```

Error: cancelled not in Status.

ID that could be string or number:

```
type ID = string | number

function findUser(id: ID) {
 console.log("Finding user: " + id)
}

findUser("abc123")
findUser(123)
```

INTERSECTION TYPES

Combine multiple types into one:

```
type Named = {
 name: string;
}

type Aged = {
 age: number;
}

type Person = Named & Aged

let person: Person = {
 name: "Ali",
 age: 25
}
```

Person has all properties from both types.

INTERSECTION WITH INTERFACES

Mix types and interfaces:

```
interface HasName {
 name: string;
}

interface HasAge {
 age: number;
}
```

```
type Person = HasName & HasAge

let person: Person = {
 name: "Ali",
 age: 25
}
```

LITERAL TYPES

Exact values only:

```
type Direction = "up" | "down" | "left" | "right"

let move: Direction = "up"
move = "diagonal"
```

Error: diagonal not in Direction.

Number literals:

```
type Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

let num: Digit = 5
num = 10
```

Error: 10 not in Digit.

Boolean literal:

```
type Yes = true

let confirmed: Yes = true
confirmed = false
```

Error: false not assignable to Yes.

FUNCTION TYPE ALIASES

Name a function type:

```
type GreetFunction = (name: string) => string

let greet: GreetFunction = function(name) {
 return "Hello " + name
}

let shout: GreetFunction = function(name) {
 return "HEY " + name.toUpperCase()
}
```

CALLBACK TYPE ALIASES

Common callback patterns:

```
type Callback = (error: Error | null, result: string) => void

function fetchData(callback: Callback) {
 callback(null, "Data loaded")
}

fetchData(function(error, result) {
 if (error) {
 console.log("Error: " + error.message)
 } else {
 console.log("Result: " + result)
 }
})
```

TUPLE TYPE ALIASES

Named tuple types:

```
type Coordinate = [number, number]

let point: Coordinate = [10, 20]

type NameAge = [string, number]

let person: NameAge = ["Ali", 25]
```

ARRAY TYPE ALIASES

Named array types:

```
type NumberList = number[]

let scores: NumberList = [95, 88, 92]

type StringList = Array<string>

let names: StringList = ["Ali", "Sara", "Omar"]
```

GENERIC TYPE ALIASES

Types with parameters:

```
type Container<T> = {
 value: T;
}
```

```
let numberBox: Container<number> = { value: 42 }
let stringBox: Container<string> = { value: "Hello" }
```

Multiple parameters:

```
type Pair<T, U> = {
 first: T;
 second: U;
}

let pair: Pair<string, number> = {
 first: "Ali",
 second: 25
}
```

CONDITIONAL TYPE BASICS

Type depends on condition:

```
type IsString<T> = T extends string ? true : false

type A = IsString<string>
type B = IsString<number>
```

A is true.

B is false.

TEMPLATE LITERAL TYPES

String types with patterns:

```
type Greeting = `Hello ${string}`
```

```
let hi: Greeting = "Hello Ali"
let bye: Greeting = "Goodbye Ali"
```

Error: Goodbye does not match pattern.

Combine with unions:

```
type Color = "red" | "blue" | "green"
type Size = "small" | "medium" | "large"

type ColorSize = `${Color}-${Size}`

let item: ColorSize = "red-small"
let wrong: ColorSize = "yellow-small"
```

Error: yellow not in Color.

EXTRACTING TYPES

Get type from existing value:

```
let user = {
 name: "Ali",
 age: 25,
 email: "ali@example.com"
}

type User = typeof user
```

User is { name: string; age: number; email: string }.

From array:

```
let colors = ["red", "green", "blue"] as const
```

type Color = typeof colors[number]

Color is "red" | "green" | "blue".

TYPE ALIAS VS INTERFACE

Both work for objects:

type UserType = { name: string }
interface UserInterface { name: string }

Type alias unique features:

Can define primitives, unions, tuples.

type ID = string | number

type Status = "active" | "inactive"

type Point = [number, number]

Interface unique features:

Can be merged (declaration merging).

Can be extended with extends keyword.

Use type alias for:

Unions and intersections.

Primitives and tuples.

Function types.

Use interface for:

Object shapes.

Class contracts.

Library type extensions.

RECURSIVE TYPES

Type that references itself:

```
type TreeNode = {
 value: string;
 children: TreeNode[];
}

let tree: TreeNode = {
 value: "root",
 children: [
 { value: "child1", children: [] },
 { value: "child2", children: [
 { value: "grandchild", children: [] }
 ]}
 ]
}
```

JSON type:

```
type JSONValue =
 | string
 | number
```

```
 | boolean
 | null
 | JSONValue[]
 | { [key: string]: JSONValue }

let data: JSONValue = {
 name: "Ali",
 scores: [95, 88, 92],
 active: true
}
```

COMMON TYPE ALIAS PATTERNS

Nullable:

```
type Nullable<T> = T | null

let user: Nullable<User> = null
```

Optional:

```
type Optional<T> = T | undefined
```

Result:

```
type Result<T, E> = { success: true; value: T } | { success: false;
error: E }
```

Dictionary:

```
type Dictionary<T> = { [key: string]: T }

let scores: Dictionary<number> = {
```

```
 Ali: 95,
 Sara: 88
}
```

PRACTICE

Create type aliases for primitives.

Define union types for status values.

Create intersection types.

Define function type aliases.

Use generic type aliases.

Practice literal and template literal types.

WHAT YOU LEARNED

You learned to create type aliases.

You learned union and intersection types.

You learned literal types.

You learned generic type aliases.

You learned when to use types vs interfaces.

The next chapter teaches functions.

BOOK 19: TYPESCRIPT BASICS
Chapter 6: Functions

FUNCTION TYPE SYNTAX

Functions have typed parameters and return:

```
function add(a: number, b: number): number {
 return a + b
}
```

Parameters have types.

Return type after colon.

ARROW FUNCTIONS

Same typing applies:

```
let add = (a: number, b: number): number => {
 return a + b
}
```

Or shorter:

```
let add = (a: number, b: number): number => a + b
```

INFERRED RETURN TYPE

TypeScript can infer return type:

```
function add(a: number, b: number) {
 return a + b
```

}

TypeScript knows it returns number.

Best practice is to add explicit return types:

```
function add(a: number, b: number): number {
 return a + b
}
```

Makes intent clear.

Catches mistakes.

VOID RETURN TYPE

Function returns nothing:

```
function logMessage(message: string): void {
 console.log(message)
}
```

No return statement.

Or empty return:

```
function logMessage(message: string): void {
 console.log(message)
 return
}
```

OPTIONAL PARAMETERS

May or may not be provided:

```
function greet(name: string, greeting?: string): string {
 if (greeting) {
 return greeting + " " + name
 }
 return "Hello " + name
}

greet("Ali")
greet("Ali", "Welcome")
```

Optional parameters are undefined if not provided.

DEFAULT PARAMETERS

Default value if not provided:

```
function greet(name: string, greeting: string = "Hello"): string {
 return greeting + " " + name
}

greet("Ali")
greet("Ali", "Welcome")
```

Type inferred from default value.

REST PARAMETERS

Accept many arguments:

```
function sum(...numbers: number[]): number {
 let total = 0
```

```
 for (let num of numbers) {
 total += num
 }
 return total
}

sum(1, 2, 3)
sum(1, 2, 3, 4, 5)
```

Rest parameter is array type.

DESTRUCTURED PARAMETERS

Object destructuring:

```
function greet({ name, age }: { name: string; age: number }): string
{
 return name + " is " + age + " years old"
}

greet({ name: "Ali", age: 25 })
```

With type alias:

```
type Person = { name: string; age: number }

function greet({ name, age }: Person): string {
 return name + " is " + age + " years old"
}
```

Array destructuring:

```
function getFirst([first]: [string, ...string[]]): string {
 return first
```

```
}

getFirst(["Ali", "Sara", "Omar"])
```

FUNCTION OVERLOADS

Multiple signatures for same function:

```
function combine(a: string, b: string): string
function combine(a: number, b: number): number
function combine(a: any, b: any): any {
 return a + b
}

let text = combine("Hello", "World")
let num = combine(5, 3)
```

First two are overload signatures.

Last is implementation.

TypeScript picks correct signature:

```
combine("Hello", 3)
```

Error: No overload matches.

THIS PARAMETER

Specify type of this:

```
type User = {
 name: string;
```

```
 greet(this: User): string;
}

let user: User = {
 name: "Ali",
 greet() {
 return "Hello " + this.name
 }
}
```

this parameter is not a real parameter.

Just tells TypeScript the this type.

FUNCTION TYPE EXPRESSIONS

Type for a function variable:

```
let greet: (name: string) => string

greet = function(name) {
 return "Hello " + name
}
```

With type alias:

```
type GreetFunction = (name: string) => string

let greet: GreetFunction = function(name) {
 return "Hello " + name
}
```

CALL SIGNATURES

Object type with call signature:

```
type GreetFunction = {
 (name: string): string;
 description: string;
}

let greet: GreetFunction = function(name) {
 return "Hello " + name
} as GreetFunction

greet.description = "A greeting function"

greet("Ali")
console.log(greet.description)
```

CONSTRUCT SIGNATURES

Type for constructor functions:

```
type PersonConstructor = {
 new (name: string): Person;
}

interface Person {
 name: string;
}

class PersonClass implements Person {
 constructor(public name: string) {}
}

function createPerson(ctor: PersonConstructor, name: string):
Person {
 return new ctor(name)
}
```

```
let person = createPerson(PersonClass, "Ali")
```

GENERIC FUNCTIONS

Functions with type parameters:

```
function identity<T>(value: T): T {
 return value
}

let num = identity<number>(42)
let str = identity<string>("Hello")
```

T is type parameter.

Caller provides actual type.

Type inference:

```
let num = identity(42)
let str = identity("Hello")
```

TypeScript infers T from argument.

MULTIPLE TYPE PARAMETERS

More than one generic type:

```
function pair<T, U>(first: T, second: U): [T, U] {
 return [first, second]
}
```

```
let result = pair<string, number>("Ali", 25)
let inferred = pair("Ali", 25)
```

GENERIC CONSTRAINTS

Limit what types are allowed:

```
function getLength<T extends { length: number }>(item: T):
number {
 return item.length
}

getLength("Hello")
getLength([1, 2, 3])
getLength({ length: 10 })
getLength(42)
```

Error: number has no length property.

USING TYPE PARAMETERS IN CONSTRAINTS

One type constrains another:

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
 return obj[key]
}

let user = { name: "Ali", age: 25 }

getProperty(user, "name")
getProperty(user, "age")
getProperty(user, "email")
```

Error: email is not a key of user.

FUNCTION COMPOSITION

Higher order function:

```
function map<T, U>(arr: T[], fn: (item: T) => U): U[] {
 return arr.map(fn)
}

let numbers = [1, 2, 3, 4]
let doubled = map(numbers, n => n * 2)
let strings = map(numbers, n => n.toString())
```

Curried function:

```
function createMultiplier(factor: number): (n: number) => number
{
 return function(n) {
 return n * factor
 }
}

let double = createMultiplier(2)
let triple = createMultiplier(3)

double(5)
triple(5)
```

CALLBACK FUNCTIONS

Function as parameter:

```
type Callback<T> = (result: T) => void

function fetchData<T>(url: string, callback: Callback<T>): void
{
 console.log("Fetching " + url)
}
```

Error callback pattern:

```
type ErrorCallback<T> = (error: Error | null, result: T | null) =>
void

function fetchData<T>(url: string, callback: ErrorCallback<T>):
void {
 callback(null, {} as T)
}

fetchData<string>("/api/data", function(error, result) {
 if (error) {
 console.log("Error: " + error.message)
 } else {
 console.log("Result: " + result)
 }
})
```

ASYNC FUNCTIONS

Return Promise:

```
async function fetchUser(id: string): Promise<User> {
 let response = await fetch("/api/users/" + id)
 let user = await response.json()
 return user
}
```

Return type is Promise<T>.

With type alias:

```
type User = { name: string; age: number }

async function getUsers(): Promise<User[]> {
 let response = await fetch("/api/users")
 return response.json()
}
```

PRACTICE

Write functions with typed parameters.

Add return types.

Use optional and default parameters.

Create function overloads.

Write generic functions.

Use constraints on generics.

WHAT YOU LEARNED

You learned function type syntax.

You learned optional and default parameters.

You learned function overloads.

You learned generic functions.

You learned constraints and higher order functions.

The next chapter teaches working with arrays.

BOOK 19: TYPESCRIPT BASICS
Chapter 7: Arrays and Tuples

ARRAY TYPES

Two syntaxes for arrays:

```
let numbers: number[] = [1, 2, 3, 4, 5]
let names: string[] = ["Ali", "Sara", "Omar"]
```

Generic syntax:

```
let numbers: Array<number> = [1, 2, 3, 4, 5]
let names: Array<string> = ["Ali", "Sara", "Omar"]
```

Both are equivalent.

First syntax is more common.

ARRAY TYPE INFERENCE

TypeScript infers array types:

```
let numbers = [1, 2, 3]
```

TypeScript knows this is number[].

let mixed = [1, "hello", true]

TypeScript infers (number | string | boolean)[].

READONLY ARRAYS

Arrays that cannot be modified:

let numbers: readonly number[] = [1, 2, 3]

numbers.push(4)

Error: push does not exist on readonly array.

Using ReadonlyArray:

let numbers: ReadonlyArray<number> = [1, 2, 3]

Same result.

Cannot add, remove, or change elements.

ARRAY METHODS WITH TYPES

Push returns new length:

let numbers: number[] = [1, 2, 3]
let length: number = numbers.push(4)

Map transforms elements:

```
let numbers: number[] = [1, 2, 3]
let doubled: number[] = numbers.map(n => n * 2)
```

TypeScript infers return type.

Filter keeps some elements:

```
let numbers: number[] = [1, 2, 3, 4, 5]
let even: number[] = numbers.filter(n => n % 2 === 0)
```

Find returns element or undefined:

```
let numbers: number[] = [1, 2, 3, 4, 5]
let found: number | undefined = numbers.find(n => n > 3)
```

Reduce combines elements:

```
let numbers: number[] = [1, 2, 3, 4, 5]
let sum: number = numbers.reduce((acc, n) => acc + n, 0)
```

TYPED CALLBACKS IN ARRAY METHODS

Map with explicit types:

```
let numbers: number[] = [1, 2, 3]

let strings: string[] = numbers.map((n: number): string => {
 return n.toString()
})
```

Filter with type guard:

```
type User = { name: string; age?: number }

let users: User[] = [
 { name: "Ali", age: 25 },
 { name: "Sara" },
 { name: "Omar", age: 30 }
]

let withAge = users.filter((user): user is User & { age: number }
=> {
 return user.age !== undefined
})
```

withAge is now (User & { age: number })[].

ARRAY OF OBJECTS

Define object shape:

```
type User = {
 name: string;
 age: number;
}

let users: User[] = [
 { name: "Ali", age: 25 },
 { name: "Sara", age: 30 }
]
```

Inline:

```
let users: { name: string; age: number }[] = [
 { name: "Ali", age: 25 },
 { name: "Sara", age: 30 }
]
```

NESTED ARRAYS

Arrays inside arrays:

```
let matrix: number[][] = [
 [1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]
]

let first: number = matrix[0][0]
```

Three dimensional:

```
let cube: number[][][] = [
 [[1, 2], [3, 4]],
 [[5, 6], [7, 8]]
]
```

TUPLE TYPES

Fixed length array with specific types:

```
let person: [string, number] = ["Ali", 25]
```

First element must be string.

Second must be number.

Accessing elements:

```
let name: string = person[0]
```

let age: number = person[1]

TUPLE WITH LABELS

Named tuple elements:

let person: [name: string, age: number] = ["Ali", 25]

Labels are for documentation only.

Access still by index.

OPTIONAL TUPLE ELEMENTS

Some elements optional:

let address: [string, string, string?] = ["123 Main", "Cairo"]

Third element is optional.

let address: [string, string, string?] = ["123 Main", "Cairo", "Egypt"]

Both are valid.

REST ELEMENTS IN TUPLES

Variable length at end:

type StringNumberBooleans = [string, number, ...boolean[]]

```
let a: StringNumberBooleans = ["hello", 1]
let b: StringNumberBooleans = ["hello", 1, true]
let c: StringNumberBooleans = ["hello", 1, true, false, true]
```

First two elements fixed.

Rest are booleans.

READONLY TUPLES

Cannot modify:

```
let point: readonly [number, number] = [10, 20]

point[0] = 5
```

Error: Cannot assign to readonly tuple.

TUPLE DESTRUCTURING

Extract values:

```
let person: [string, number] = ["Ali", 25]

let [name, age] = person

console.log(name)
console.log(age)
```

TUPLE IN FUNCTION RETURNS

Return multiple values:

```
function getMinMax(numbers: number[]): [number, number] {
 let min = Math.min(...numbers)
 let max = Math.max(...numbers)
 return [min, max]
}

let [min, max] = getMinMax([1, 5, 3, 9, 2])
```

ARRAY VS TUPLE

Array:

Variable length.

All elements same type.

```
let numbers: number[] = [1, 2, 3, 4, 5]
```

Tuple:

Fixed length.

Each position has specific type.

```
let person: [string, number, boolean] = ["Ali", 25, true]
```

SPREAD WITH ARRAYS

Spread operator:

```
let arr1: number[] = [1, 2, 3]
let arr2: number[] = [4, 5, 6]

let combined: number[] = [...arr1, ...arr2]
```

TypeScript preserves type.

SPREAD WITH TUPLES

Spread in tuple:

```
let tuple1: [number, number] = [1, 2]
let tuple2: [number, number] = [3, 4]

let combined: [number, number, number, number] = [...tuple1,
...tuple2]
```

CONST ASSERTIONS

Narrow to literal types:

```
let numbers = [1, 2, 3] as const
```

Type is readonly [1, 2, 3] not number[].

```
let person = ["Ali", 25] as const
```

Type is readonly ["Ali", 25].

Useful for enums:

```
let COLORS = ["red", "green", "blue"] as const
```

type Color = typeof COLORS[number]

Color is "red" | "green" | "blue".

ARRAY TYPE GUARDS

Check if array:

```
function processData(data: string | string[]) {
 if (Array.isArray(data)) {
 data.forEach(item => console.log(item))
 } else {
 console.log(data)
 }
}
```

Check array contents:

```
function isStringArray(arr: unknown[]): arr is string[] {
 return arr.every(item => typeof item === "string")
}

let mixed: unknown[] = ["a", "b", "c"]

if (isStringArray(mixed)) {
 mixed.forEach(s => console.log(s.toUpperCase()))
}
```

COMMON PATTERNS

Empty array with type:

```
let users: User[] = []
```

Array from function:

```
function getNumbers(): number[] {
 return [1, 2, 3, 4, 5]
}
```

Generic array function:

```
function first<T>(arr: T[]): T | undefined {
 return arr[0]
}

let num = first([1, 2, 3])
let str = first(["a", "b", "c"])
```

PRACTICE

Create typed arrays.

Use array methods with proper types.

Define and use tuples.

Use readonly arrays and tuples.

Practice const assertions.

Write functions returning tuples.

WHAT YOU LEARNED

You learned array type syntax.

You learned array methods with types.

You learned tuple types and labels.

You learned readonly and const assertions.

You learned spread with arrays and tuples.

The next chapter teaches TypeScript programs.
BOOK 19: TYPESCRIPT BASICS
Chapter 8: TypeScript Programs

This chapter contains complete TypeScript programs.

Each program demonstrates concepts from previous chapters.

Study these programs to understand how pieces fit together.

PROGRAM 1: TODO LIST

A typed todo list with CRUD operations.

Demonstrates interfaces, arrays, functions.

todo.ts:

```
interface Todo {
```

```
 id: number;
 title: string;
 completed: boolean;
 createdAt: Date;
}

type TodoList = Todo[]

let todos: TodoList = []
let nextId: number = 1

function addTodo(title: string): Todo {
 let todo: Todo = {
 id: nextId++,
 title: title,
 completed: false,
 createdAt: new Date()
 }
 todos.push(todo)
 return todo
}

function removeTodo(id: number): boolean {
 let index = todos.findIndex(todo => todo.id === id)
 if (index !== -1) {
 todos.splice(index, 1)
 return true
 }
 return false
}

function toggleTodo(id: number): Todo | undefined {
 let todo = todos.find(todo => todo.id === id)
 if (todo) {
 todo.completed = !todo.completed
 }
 return todo
}

function getTodos(filter?: "all" | "completed" | "pending"): TodoList {
```

```
 if (!filter || filter === "all") {
 return todos
 }
 if (filter === "completed") {
 return todos.filter(todo => todo.completed)
 }
 return todos.filter(todo => !todo.completed)
}

function clearCompleted(): number {
 let before = todos.length
 todos = todos.filter(todo => !todo.completed)
 return before - todos.length
}

addTodo("Learn TypeScript")
addTodo("Build a project")
addTodo("Practice daily")

toggleTodo(1)

console.log("All todos:")
getTodos().forEach(todo => {
 let status = todo.completed ? "[X]" : "[ ]"
 console.log(status + " " + todo.title)
})

console.log("\nPending todos:")
getTodos("pending").forEach(todo => {
 console.log("[ ] " + todo.title)
})
```

PROGRAM 2: USER MANAGEMENT

User system with authentication.

Demonstrates type aliases, unions, optional properties.

user.ts:

```
type UserRole = "admin" | "editor" | "viewer"

interface User {
 id: string;
 email: string;
 name: string;
 role: UserRole;
 createdAt: Date;
 lastLogin?: Date;
}

interface AuthResult {
 success: boolean;
 user?: User;
 error?: string;
}

type UserDatabase = Map<string, User & { passwordHash: string }
>

let users: UserDatabase = new Map()

function generateId(): string {
 return Math.random().toString(36).substring(2, 9)
}

function hashPassword(password: string): string {
 return "hashed_" + password
}

function register(
 email: string,
 password: string,
 name: string,
 role: UserRole = "viewer"
): AuthResult {
```

```
 if (findUserByEmail(email)) {
 return { success: false, error: "Email already exists" }
 }

 let user = {
 id: generateId(),
 email: email,
 name: name,
 role: role,
 createdAt: new Date(),
 passwordHash: hashPassword(password)
 }

 users.set(user.id, user)

 let publicUser: User = {
 id: user.id,
 email: user.email,
 name: user.name,
 role: user.role,
 createdAt: user.createdAt
 }

 return { success: true, user: publicUser }
}

function findUserByEmail(email: string): User | undefined {
 for (let user of users.values()) {
 if (user.email === email) {
 let publicUser: User = {
 id: user.id,
 email: user.email,
 name: user.name,
 role: user.role,
 createdAt: user.createdAt,
 lastLogin: user.lastLogin
 }
 return publicUser
 }
 }
```

```
 return undefined
}

function login(email: string, password: string): AuthResult {
 for (let user of users.values()) {
 if (user.email === email) {
 if (user.passwordHash === hashPassword(password)) {
 user.lastLogin = new Date()
 let publicUser: User = {
 id: user.id,
 email: user.email,
 name: user.name,
 role: user.role,
 createdAt: user.createdAt,
 lastLogin: user.lastLogin
 }
 return { success: true, user: publicUser }
 }
 return { success: false, error: "Invalid password" }
 }
 }
 return { success: false, error: "User not found" }
}

function hasPermission(user: User, action: string): boolean {
 let permissions: Record<UserRole, string[]> = {
 admin: ["read", "write", "delete", "manage"],
 editor: ["read", "write"],
 viewer: ["read"]
 }
 return permissions[user.role].includes(action)
}

let result1 = register("ali@example.com", "secret123", "Ali",
"admin")
console.log("Register:", result1)

let result2 = login("ali@example.com", "secret123")
console.log("Login:", result2)
```

```
if (result2.success && result2.user) {
 console.log("Can delete:", hasPermission(result2.user, "delete"))
 console.log("Can manage:", hasPermission(result2.user, "manage"))
}
```

PROGRAM 3: SHOPPING CART

E-commerce cart with discounts.

Demonstrates generics, intersection types, calculations.

cart.ts:

```
interface Product {
 id: string;
 name: string;
 price: number;
 category: string;
}

interface CartItem {
 product: Product;
 quantity: number;
}

interface Discount {
 code: string;
 type: "percentage" | "fixed";
 value: number;
 minPurchase?: number;
}

interface Cart {
 items: CartItem[];
 discount?: Discount;
}
```

```
interface CartSummary {
 subtotal: number;
 discountAmount: number;
 total: number;
 itemCount: number;
}

let cart: Cart = { items: [] }

function addToCart(product: Product, quantity: number = 1): void
{
 let existing = cart.items.find(item => item.product.id ===
product.id)
 if (existing) {
 existing.quantity += quantity
 } else {
 cart.items.push({ product, quantity })
 }
}

function removeFromCart(productId: string): boolean {
 let index = cart.items.findIndex(item => item.product.id ===
productId)
 if (index !== -1) {
 cart.items.splice(index, 1)
 return true
 }
 return false
}

function updateQuantity(productId: string, quantity: number):
boolean {
 let item = cart.items.find(item => item.product.id ===
productId)
 if (item) {
 if (quantity <= 0) {
 return removeFromCart(productId)
 }
 item.quantity = quantity
```

```
 return true
 }
 return false
}

function applyDiscount(discount: Discount): boolean {
 let subtotal = calculateSubtotal()
 if (discount.minPurchase && subtotal < discount.minPurchase) {
 return false
 }
 cart.discount = discount
 return true
}

function calculateSubtotal(): number {
 return cart.items.reduce((total, item) => {
 return total + item.product.price * item.quantity
 }, 0)
}

function calculateDiscount(subtotal: number): number {
 if (!cart.discount) {
 return 0
 }
 if (cart.discount.type === "percentage") {
 return subtotal * (cart.discount.value / 100)
 }
 return Math.min(cart.discount.value, subtotal)
}

function getCartSummary(): CartSummary {
 let subtotal = calculateSubtotal()
 let discountAmount = calculateDiscount(subtotal)
 let itemCount = cart.items.reduce((count, item) => count +
item.quantity, 0)

 return {
 subtotal: subtotal,
 discountAmount: discountAmount,
 total: subtotal - discountAmount,
```

```
 itemCount: itemCount
 }
}

function clearCart(): void {
 cart.items = []
 cart.discount = undefined
}

let laptop: Product = {
 id: "1",
 name: "Laptop",
 price: 999.99,
 category: "Electronics"
}

let mouse: Product = {
 id: "2",
 name: "Mouse",
 price: 29.99,
 category: "Electronics"
}

let book: Product = {
 id: "3",
 name: "TypeScript Guide",
 price: 39.99,
 category: "Books"
}

addToCart(laptop, 1)
addToCart(mouse, 2)
addToCart(book, 1)

console.log("Cart before discount:")
console.log(getCartSummary())

let discount: Discount = {
 code: "SAVE10",
 type: "percentage",
```

```
 value: 10,
 minPurchase: 50
}

applyDiscount(discount)

console.log("\nCart after 10% discount:")
console.log(getCartSummary())
```

PROGRAM 4: API CLIENT

HTTP client with typed responses.

Demonstrates generics, async/await, error handling.

api.ts:

```
interface ApiResponse<T> {
 data: T;
 status: number;
 message: string;
}

interface ApiError {
 status: number;
 message: string;
 details?: string;
}

type Result<T> = { success: true; value: T } | { success: false;
error: ApiError }

interface User {
 id: number;
 name: string;
 email: string;
```

```
}

interface Post {
 id: number;
 userId: number;
 title: string;
 body: string;
}

class ApiClient {
 private baseUrl: string

 constructor(baseUrl: string) {
 this.baseUrl = baseUrl
 }

 async get<T>(endpoint: string): Promise<Result<T>> {
 try {
 let response = await fetch(this.baseUrl + endpoint)

 if (!response.ok) {
 return {
 success: false,
 error: {
 status: response.status,
 message: "Request failed"
 }
 }
 }

 let data = await response.json() as T
 return { success: true, value: data }
 } catch (err) {
 return {
 success: false,
 error: {
 status: 0,
 message: "Network error",
 details: String(err)
 }
```

```
 }
 }
 }

 async post<T, U>(endpoint: string, body: T):
Promise<Result<U>> {
 try {
 let response = await fetch(this.baseUrl + endpoint, {
 method: "POST",
 headers: { "Content-Type": "application/json" },
 body: JSON.stringify(body)
 })

 if (!response.ok) {
 return {
 success: false,
 error: {
 status: response.status,
 message: "Request failed"
 }
 }
 }

 let data = await response.json() as U
 return { success: true, value: data }
 } catch (err) {
 return {
 success: false,
 error: {
 status: 0,
 message: "Network error",
 details: String(err)
 }
 }
 }
 }
}

async function main() {
 let api = new ApiClient("https://jsonplaceholder.typicode.com")
```

```
 let usersResult = await api.get<User[]>("/users")

 if (usersResult.success) {
 console.log("Users loaded:", usersResult.value.length)
 usersResult.value.slice(0, 3).forEach(user => {
 console.log("- " + user.name + " (" + user.email + ")")
 })
 } else {
 console.log("Error:", usersResult.error.message)
 }

 let postResult = await api.get<Post>("/posts/1")

 if (postResult.success) {
 console.log("\nPost:", postResult.value.title)
 }

 type NewPost = Omit<Post, "id">

 let newPost: NewPost = {
 userId: 1,
 title: "My TypeScript Post",
 body: "TypeScript makes JavaScript better"
 }

 let createResult = await api.post<NewPost, Post>("/posts",
newPost)

 if (createResult.success) {
 console.log("\nCreated post with ID:", createResult.value.id)
 }
}

main()
```

PROGRAM 5: EVENT SYSTEM

Typed event emitter.

Demonstrates generics, callbacks, type safety.

events.ts:

```
type EventCallback<T> = (data: T) => void

interface EventMap {
 [event: string]: unknown;
}

class TypedEventEmitter<Events extends EventMap> {
 private listeners: Map<keyof Events,
Set<EventCallback<unknown>>>

 constructor() {
 this.listeners = new Map()
 }

 on<K extends keyof Events>(
 event: K,
 callback: EventCallback<Events[K]>
 ): void {
 if (!this.listeners.has(event)) {
 this.listeners.set(event, new Set())
 }
 this.listeners.get(event)!.add(callback as
EventCallback<unknown>)
 }

 off<K extends keyof Events>(
 event: K,
 callback: EventCallback<Events[K]>
 ): void {
 let eventListeners = this.listeners.get(event)
 if (eventListeners) {
 eventListeners.delete(callback as EventCallback<unknown>)
```

```
 }
 }

 emit<K extends keyof Events>(event: K, data: Events[K]): void {
 let eventListeners = this.listeners.get(event)
 if (eventListeners) {
 eventListeners.forEach(callback => callback(data))
 }
 }

 once<K extends keyof Events>(
 event: K,
 callback: EventCallback<Events[K]>
 ): void {
 let wrapper = ((data: Events[K]) => {
 callback(data)
 this.off(event, wrapper)
 }) as EventCallback<Events[K]>
 this.on(event, wrapper)
 }
}

interface AppEvents {
 userLoggedIn: { userId: string; timestamp: Date };
 userLoggedOut: { userId: string };
 messageReceived: { from: string; content: string };
 error: { code: number; message: string };
}

let events = new TypedEventEmitter<AppEvents>()

events.on("userLoggedIn", data => {
 console.log("User logged in:", data.userId, "at", data.timestamp)
})

events.on("messageReceived", data => {
 console.log("Message from", data.from + ":", data.content)
})

events.on("error", data => {
```

```
 console.error("Error " + data.code + ":", data.message)
})

events.once("userLoggedIn", data => {
 console.log("First login detected for:", data.userId)
})

events.emit("userLoggedIn", {
 userId: "user123",
 timestamp: new Date()
})

events.emit("messageReceived", {
 from: "Ali",
 content: "Hello TypeScript!"
})

events.emit("error", {
 code: 404,
 message: "Resource not found"
})

events.emit("userLoggedIn", {
 userId: "user456",
 timestamp: new Date()
})
```

WHAT YOU LEARNED

You learned to build complete TypeScript programs.

You learned to combine types, interfaces, and functions.

You learned practical type patterns.

You have reference code for your projects.

BOOK COMPLETE

You have completed Book 19: TypeScript Basics.

You learned the fundamentals of TypeScript.

You can add types to variables and functions.

You can create interfaces and type aliases.

You understand arrays, tuples, and generics.

Continue to Book 20 for advanced TypeScript.